Rivet user manual

Andy Buckley[a], Jonathan Butterworth[b], David Grellscheid[c], Hendrik Hoeth[c], Leif Lönnblad[d], James Monk[b], Holger Schulz[e], Frank Siegert[f,*]

[a] *PPE Group, School of Physics, University of Edinburgh, UK.*
[b] *HEP Group, Dept. of Physics and Astronomy, UCL, London, UK.*
[c] *IPPP, Durham University, UK.*
[d] *Theoretical Physics, Lund University, Sweden.*
[e] *Institut für Physik, Berlin Humboldt University, Germany.*
[f] *Physikalisches Institut, Freiburg University, Germany.*

Abstract

This is the manual and user guide for the Rivet system for the validation and tuning of Monte Carlo event generators. As well as the core Rivet library, this manual describes the usage of the `rivet` program and the AGILe generator interface library. The depth and level of description is chosen for users of the system, starting with the basics of using validation code written by others, and then covering sufficient details to write new Rivet analyses and calculational components.

Keywords: Event generator; simulation; validation; tuning; QCD

PROGRAM SUMMARY

Manuscript Title: Rivet user manual
Authors: Andy Buckley, Jonathan Butterworth, David Grellscheid, Hendrik Hoeth, Leif Lönnblad, James Monk, Holger Schulz, Frank Siegert
Program Title: Rivet
Journal Reference:
Catalogue identifier:
Licensing provisions:
Programming language: C++, Python
Computer: PC running Linux, Mac
Operating system: Linux, Mac OS

*Corresponding author.
E-mail address: frank.siegert@cern.ch

RAM: 20M bytes
Number of processors used: 1
Supplementary material:
Keywords: Event generator, simulation, validation, tuning, QCD
Classification: 11.9 Event Reconstruction and Data Analysis
External routines/libraries: HepMC, GSL, FastJet, Python, Swig, Boost, YAML
Nature of problem:
Experimental measurements from high-energy particle colliders should be defined and stored in a general framework such that it is simple to compare theory predictions to them. Rivet is such a framework, and contains at the same time a large collection of existing measurements.
Solution method:
Rivet is based on HepMC events, a standardised output format provided by many theory simulation tools. Events are processed by Rivet to generate histograms for the requested list of analyses, incorporating all experimental phase space cuts and histogram definitions.
Restrictions:
Can not calculate statistical errors for correlated events as they appear in NLO calculations.
Unusual features:
It is possible for the user to implement and use their own custom analysis as a module without having to modify the main Rivet code/installation.
Additional comments:

Running time:
Depends on the number and complexity of analyses being applied, but typically a few hundred events per second.

1. Introduction

This manual is a users' guide to using the Rivet generator validation system. Rivet is a C++ class library, which provides the infrastructure and calculational tools for particle-level analyses for high energy collider experiments, enabling physicists to validate event generator models and tunings with minimal effort and maximum portability. Rivet is designed to scale effectively to large numbers of analyses for truly global validation, by transparent use of an automated result caching system.

The Rivet ethos, if it may be expressed succinctly, is that user analysis code should be extremely clean and easy to write — ideally it should be

sufficiently self-explanatory to in itself be a reference to the experimental analysis algorithm — without sacrificing power or extensibility. The machinery to make this possible is intentionally hidden from the view of all but the most prying users. Generator independence is explicitly required by virtue of all analyses operating on the generic "HepMC" event record.

The simplest way to use Rivet is via the `rivet` command line tool, which analyses textual HepMC event records as they are generated and produces output distributions in a structured textual format. The input events are generated using the generator's own steering program, if one is provided; for generators which provide no default way to produce HepMC output, the AGILe generator interface library, and in particular the `agile-runmc` command which it provides, may be useful. For those who wish to embed their analyses in some larger framework, Rivet can also be used as a library to run programmatically on HepMC event objects with no special executable being required.

Before we get started, a declaration of intent: this manual is intended to be a guide to using Rivet, rather than a comprehensive and painstakingly maintained reference to the application programming interface (API) of the Rivet library. For that purpose the online documentation at `http://rivet.hepforge.org` should be sufficient – in case of confusion please contact the authors at `rivet@projects.hepforge.org`. Similar API documentation is maintained for AGILe at `http://agile.hepforge.org`.

1.1. Typographic conventions

As is normal in computer user manuals, the typography in this manual is used to indicate whether we are describing source code elements, commands to be run in a terminal, the output of a command etc.

The main such clue will be the use of `typewriter-style` text: this indicates the name of a command or code element — class names, function names etc. Typewriter font is also used for commands to be run in a terminal, but in this case it will be prefixed by a dollar sign, as in `$ echo "Hello" | cat`. The output of such a command on the terminal will be typeset in sans-serif font. When we are documenting a code feature in detail (which is not the main point of this manual), we will use square brackets to indicate optional arguments, and italic font between angle brackets to represent an argument name which should be replaced by a value, e.g. `Event::applyProjection(`⟨*proj*⟩`)`.

Part I
Getting started with Rivet

As with many things, Rivet may be meaningfully approached at several distinct levels of detail:

- The simplest, and we hope the most common, is to use the analyses which are already in the library to study events from a variety of generators and tunes: this is enormously valuable in itself and we encourage all manner of experimentalists and phenomenologists alike to use Rivet in this mode.

- A more involved level of usage is to write your own Rivet analyses — this may be done without affecting the installed standard analyses by use of a "plugin" system (although we encourage users who develop analyses to submit them to the Rivet developers for inclusion into a future release of the main package). This approach requires some understanding of programming within Rivet but you don't *need* to know about exactly what the system is doing with the objects that you have defined.

- Finally, Rivet developers and people who want to do non-standard things with their analyses will need to know something about the messy details of what Rivet's infrastructure is doing behind the scenes. But you'd probably rather be doing some physics!

The current part of this manual is for the first sort of user, who wants to get on with studying some observables with a generator or tune, or comparing several such models. Since everyone will fall into this category at some point, our present interest is to get you to that all-important "physics plots" stage as quickly as possible. Analysis authors and Rivet service-mechanics will find the more detailed information that they crave in Part III.

2. Quickstart

The point of this section is to get you up and running with Rivet as soon as possible. Doing this by hand may be rather frustrating, as Rivet depends on several external libraries — you'll get bored downloading and building them by hand in the right order. Here we recommend a much simpler way —

for the full details of how to build Rivet by hand, please consult the Rivet Web page.

Bootstrap script. We have written a bootstrapping script which will download tarballs of Rivet, AGILe and the other required libraries, expand them and build them in the right order with the correct build flags. This is generally nicer than doing it all by hand, and virtually essential if you want to use the existing versions of FastJet, HepMC, generator libraries, and so on from CERN AFS: there are issues with these versions which the script works around, which you won't find easy to do yourself.

To run the script, we recommend that you choose a personal installation directory, i.e. make a `~/local` directory for this purpose, to avoid polluting your home directory with a lot of files. If you already use a directory of the same name, you might want to use a separate one, say `~/rivetlocal`, such that if you need to delete everything in the installation area you can do so without difficulties.

Now, change directory to your build area (you may also want to make this, e.g. `~/build`), and download the script:

```
$ wget http://rivet.hepforge.org/svn/bootstrap/rivet-bootstrap
$ chmod +x rivet-bootstrap
```

Now run it to get some help: `$ ./rivet-bootstrap --help`
Now to actually do the install: for example, to bootstrap Rivet and AGILe to the install area specified as the prefix argument, run this:

```
$ ./rivet-bootstrap --install-agile --prefix=⟨localdir⟩
```

If you are running on a system where the CERN AFS area is mounted as `/afs/cern.ch`, then the bootstrap script will attempt to use the pre-built HepMC[1], LHAPDF[2], FastJet[3, 4] and GSL libraries from the LCG software area. Either way, finally the bootstrap script will write out a file containing the environment settings which will make the system useable. You can source this file, e.g. `source rivetenv.sh` to make your current shell ready-to-go for a Rivet run (use `rivetenv.csh` if you are a C shell user).

You now have a working, installed copy of the Rivet and AGILe libraries, and the `rivet` and `agile-runmc` executables: respectively these are the command-line frontend to the Rivet analysis library, and a convenient steering command for generators which do not provide their own main program with HepMC output. To test that they work as expected, source the setup scripts as above, if you've not already done so, and run this:

```
$ rivet --help
```

This should print a quick-reference user guide for the `rivet` command to the terminal. Similarly, for `agile-runmc`,

```
$ agile-runmc --help
$ agile-runmc --list-gens
$ agile-runmc --beams=pp:14000 Pythia6:425
```

which should respectively print the help, list the available generators and make 10 LHC-type events using the Fortran Pythia[5] 6.423 generator. You're on your way! If no generators are listed, you probaby need to install a local Genser-type generator repository: see section 2.1.

In this manual, because of its convenience, we will use `agile-runmc` as our canonical way of producing a stream of HepMC event data; if your interest is in running a generator like Sherpa[6], Pythia 8[7, 8], or Herwig++[9] which provides their own native way to make HepMC output, or a generator like PHOJET which is not currently supported by AGILe, then substitute the appropriate command in what follows. We'll discuss using these commands in detail in section 3.

2.1. Getting generators for AGILe

One last thing before continuing, though: the generators themselves. Again, if you're running on a system with the CERN LCG AFS area mounted, then `agile-runmc` will attempt to automatically use the generators packaged by the LCG Genser team.

Otherwise, you'll have to build your own mirror of the LCG generators. This process is evolving with time, and so, rather than provide information in this manual which will be outdated by the time you read it, we simply refer you to the relevant page on the Rivet wiki: `http://rivet.hepforge.org/trac/wiki/GenserMirror`.

If you are interested in using a generator not currently supported by AGILe, which does not output HepMC events in its native state, then please contact the authors (via the Rivet developer contact email address) and hopefully we can help.

2.2. Command completion

A final installation point worth considering is using the supplied bash-shell programmable completion setup for the `rivet` and `agile-runmc` commands. Despite being cosmetic and semi-trivial, programmable completion makes using `rivet` positively pleasant, especially since you no longer need to remember

the somewhat cryptic analysis names[1]!

To use programmable completion, source the appropriate files from the install location:

```
$ . ⟨localdir⟩/share/Rivet/rivet-completion
$ . ⟨localdir⟩/share/AGILe/agile-completion
```

(if you are using the setup script `rivetenv.sh` this is automatically done for you). If there is already a ⟨*localdir*⟩`/etc/bash_completion.d` directory in your install path, Rivet and AGILe's installation scripts will install extra copies into that location, since automatically sourcing all completion files in such a path is quite standard.

3. Running Rivet analyses

The `rivet` executable is the easiest way to use Rivet, and will be our example throughout this manual. This command reads HepMC events in the standard ASCII format, either from file or from a text stream.

3.1. The FIFO idiom

Since you rarely want to store simulated HepMC events and they are computationally cheap to produce (at least when compared to the remainder of experiment simulation chains), we recommend using a Unix *named pipe* (or "FIFO" — first-in, first-out) to stream the events. While this may seem unusual at first, it is just a nice way of "pretending" that we are writing to and reading from a file, without actually involving any slow disk access or building of huge files: a 1M event LHC run would occupy $\sim 60GB$ on disk, and typically it takes twice as long to make and analyse the events when the filesystem is involved! Here is an example:

```
$ mkfifo fifo.hepmc
$ agile-runmc Pythia6:425 -o fifo.hepmc &
$ rivet -a EXAMPLE fifo.hepmc
```

Note that the generator process (`agile-runmc` in this case) is *backgrounded* before `rivet` is run.

Notably, `mkfifo` will not work if applied to a directory mounted via the AFS distributed filesystem, as widely used in HEP. This is not a big problem: just make your FIFO object somewhere not mounted via AFS, e.g. `/tmp`.

[1] Standard Rivet analyses have names which, as well as the publication date and experiment name, incorporate the 8-digit Spires/Inspire ID code.

There is no performance penalty, as the filesystem object is not written to during the streaming process.

In the following command examples, we will assume that a generator has been set up to write to the `fifo.hepmc` FIFO, and just list the `rivet` command that reads from that location. Some typical `agile-runmc` commands are listed in appendix Appendix A.

3.2. Analysis status

The standard Rivet analyses are divided into four status classes: validated, preliminary, obsolete, and unvalidated (in roughly decreasing order of academic acceptability).

The Rivet "validation procedure" is not formally defined, but generally implies that an analysis has been checked to ensure reproduction of MC points shown in the paper where possible, and is believed to have no outstanding issues with analysis procedure or cuts. Additionally, analyses marked as "validated" and distributed with Rivet should normally have been code-checked by an experienced developer to ensure that the code is a good example of Rivet usage and is not more complex than required or otherwise difficult to read or maintain. Such analyses are regarded as fully ready for use in any MC validation or tuning studies.

Validated analyses which implement an unfinished piece of experimental work are considered to be trustworthy in their implementation of a conference note or similar "informal" publication, but do not have the magic stamp of approval that comes from a journal publication. This remains the standard mark of experimental respectability and accordingly we do not include such analyses in the Rivet standard analysis libraries, but in a special "preliminary" library. While preliminary analyses may be used for physics studies, please be aware of the incomplete status of the corresponding experimental study, and also be aware that the histograms in such analyses may be renamed or removed entirely, as may the analysis itself.

Preliminary analyses will not have a Spires/Inspire ID, and hence on their move into the standard Rivet analysis library they will normally undergo a name change: please ensure when you upgrade between Rivet versions that any scripts or programs which were using preliminary analyses are not broken by the disappearance or change of that analysis in the newer version. The minor perils of using preliminary analyses can be avoided by the cautious by building Rivet with the `--disable-preliminary` configuration flag, in which case their temptation will not even be offered.

To make transitions between Rivet versions more smooth and predictable for users of preliminary analyses, preliminary analyses which are superseded by a validated version will be reclassified as obsolete and will be retained for one major version of Rivet with a status of "obsolete" before being removed, to give users time to migrate their run scripts, i.e. if an analysis is marked as obsolete in version 1.4.2, it will remain in Rivet's distribution until version 1.5.0. Obsolete analyses may have different reference histograms from the final version and will not be maintained. Obsolete analyses will not be built if either the `--disable-obsolete` configuration flag is specified at build time: for convenience, the default value of this flag is the value of the `--disable-preliminary` flag.

Finally, unvalidated analyses are those whose implementation is incomplete, flawed or just troubled by doubts. Running such analyses is not a good idea if you aren't trying to fix them, and Rivet's command line tools will print copious warning messages if you do. Unvalidated analyses in the Rivet distribution are not built by default, as they are only of interest to developers and would be distracting clutter for the majority of users: if you *really* need them, building Rivet with the `--enable-unvalidated` configuration flag will slake your thirst for danger.

3.3. Example `rivet` commands

- **Getting help:** `rivet --help` will print a (hopefully) helpful list of options which may be used with the `rivet` command, as well as other information such as environment variables which may affect the run.

- **Choosing analyses:** `rivet --list-analyses` will list the available analyses, including both those in the Rivet distribution and any plugins which are found at runtime. `rivet --show-analysis` ⟨*patt*⟩ will show a lot of details about any analyses whose name match the ⟨*patt*⟩ regular expression pattern — simple bits of analysis name are a perfectly valid subset of this. For example, `rivet --show-analysis CDF_200` exploits the standard Rivet analysis naming scheme to show details of all available CDF experiment analyses published in the "noughties."

- **Running analyses:** `rivet -a DELPHI_1996_S3430090 fifo.hepmc` will run the Rivet `DELPHI_1996_S3430090` [10] analysis on the events in the `fifo.hepmc` file (which, from the name, is probably a filesystem named pipe rather than a normal *file*). This analysis is the one originally

used for the DELPHI "PROFESSOR" generator tuning. If the first event in the data file does not have appropriate beam particles, the analysis will be disabled; since there is only one analysis in this case, the command will exit immediately with a warning if the first event is not an e^+e^- event.

- **Histogramming:** `rivet fifo.hepmc -H foo.aida` will read all the events in the `fifo.hepmc` file. The `-H` switch is used to specify that the output histogram file will be named `foo.aida`. By default the output file is called `Rivet.aida`.

- **Fine-grained logging:**

  ```
  rivet fifo.hepmc -A -l Rivet.Analysis=DEBUG \
  -l Rivet.Projection=DEBUG -l Rivet.Projection.FinalState=TRACE \
  -l NEvt=WARN
  ```
  will analyse events as before, but will print different status information as the run progresses. Hierarchical logging control is possible down to the level of individual analyses and projections as shown above; this is useful for debugging without getting overloaded with debug information from *all* the components at once. The default level is "INFO", which lies between "DEBUG" and "WARNING"; the "TRACE" level is for very low level information, and probably isn't needed by normal users.

4. Using analysis data

In this section, we summarise how to use the data files which Rivet produces for plotting, validation and tuning.

4.1. Histogram formats

Rivet currently produces output histogram data in the AIDA XML format. Most people aren't familiar with AIDA (and we recommend that you remain that way!), and it will disappear entirely from Rivet in version 2.0. If you do not want to use the plotting tools that come with Rivet (cf. Sec. 4.4), you might wish to cast the AIDA files to a different format for plotting, and for this we supply several scripts.

Conversion to ROOT. Rivet installs an `aida2root` script, which converts the AIDA records to a `.root` file full of ROOT `TGraph`s. One word of warning: a bug in ROOT means that `TGraph`s do not render properly from file because the axis is not drawn by default. To display the plots correctly in ROOT you will need to pass the `"AP"` drawing option string to either the `TGraph::Draw()` method, or in the options box in the `TBrowser` GUI interface. Alternatively you can also use the `-t` option with which `aida2root` produces `TH1`s instead.

Conversion to "flat format". Most of our histogramming is based around a "flat" plain text format, which can easily be read (and written) by hand. We provide a script called `aida2flat` to do this conversion. Run `aida2flat -h` to get usage instructions; in particular the Gnuplot and "split output" options are useful for further visualisation. Aside from anything else, this is useful for simply checking the contents of an AIDA file, with `aida2flat Rivet.aida | less`.

We get asked a lot about why we don't use ROOT internally: aside from a general unhappiness about the design and quality of the data objects in ROOT, the monolithic nature of the system makes it a big dependency for a system as small as Rivet. While not an issue for experimentalists, most theorists and generator developers do not use ROOT and we preferred to embed the AIDA system, which in its LWH implementation requires no external package. The replacement for AIDA will be another lightweight system rather than ROOT, with an emphasis on friendly, intuitive data object design, and correct handling of sample merging statistics for all data objects.

4.2. Chopping histograms

In some cases you don't want to keep the complete histograms produced by Rivet. For generator tuning purposes, for example, you want to get rid of the bins you already know your generator is incapable of describing. You can use the script `rivet-chopbins` to specify those bin-ranges you want to keep individually for each histogram in a Rivet output-file. The bin-ranges have to be specified using the corresponding x-values of that histogram. The usage is very simple. You can specify bin ranges of histograms to keep on the command-line via the `-b` switch, which can be given multiple times, e.g. `rivet-chopbins -b /CDF_2001_S4751469/d03-x01-y01:5:13 Rivet.aida`

will chop all bins with $x < 5$ and $x > 13$ from the histogram `/CDF_2001_S4751469/d03x01y01` in the file `Rivet.aida`. (In this particular case, x would be a leading jet $p_\perp$.)

4.3. Normalising histograms

Sometimes you want to use histograms normalised to, e.g., the generator cross-section or the area of a reference-data histogram. The script `rivet-rescale` was designed for these purposes. The usage is the following:
`rivet-rescale -O observables -r RIVETDATA -o normalised Rivet.aida`
By default, the normalised histograms are written to file in the AIDA-XML format. You can also give the `-f` switch on the command line to produce flat histograms.

Normalising to reference data. You will need an output-file of Rivet, `Rivet.aida`, a folder that contains the reference-data histograms (e.g. `rivet-config --datadir`) and optionally, a text-file, `observables` that contains the names of the histograms you would like to normalise - those not given in the file will remain un-normalised. These are examples of how your `observables` file might look like:

```
/CDF_2000_S4155203/d01-x01-y01
```

If a histogram `/CDF_2000_S4155203/d01-x01-y01` is found in one of the reference-data files in the folder specified via the `-r` switch, then this will result in a histogram `/CDF_2000_S4155203/d01-x01-y01` being normalised to the area of the corresponding reference-data histogram. You can further specify a certain range of bins to normalise:

```
/CDF_2000_S4155203/d01-x01-y01:2:35
```

will chop off the bins with $x < 2$ and $x > 35$ of both, the histogram in your `Rivet.aida` and the reference-data histogram. The remaining MC histogram is then normalised to the remaining area of the reference-data histogram.

Normalising to arbitrary areas. In the file `observables` you can further specify an arbitrary number, e.g. a generator cross-section, as follows:

```
/CDF_2000_S4155203/d01-x01-y01  1.0
```

will result in the histogram `/CDF_2000_S4155203/d01-x01-y01` being normalised to 1.0, and

```
/CDF_2000_S4155203/d01-x01-y01:2:35  1.0
```

will chop off the bins with $x < 2$ and $x > 35$ of the histogram `/CDF_2000_S4155203/d01-x01-y01` first and normalise the remaining histogram to one.

4.4. Plotting and comparing data

Rivet comes with three commands — `rivet-mkhtml`, `compare-histos` and `make-plots` — for comparing and plotting data files. These commands produce nice comparison plots of publication quality from the AIDA format text files.

The high level program `rivet-mkhtml` will automatically create a plot webpage from the given AIDA files. It searches for reference data automatically and uses the other two commands internally. Example:
`$ rivet-mkhtml withUE.aida:'Title=With UE' withoutUE.aida:'LineColor=blue'`
Run `rivet-mkhtml --help` to find out about all features and options.

You can also run the other two commands separately:

- `compare-histos` will accept a number of AIDA files as input (ending in `.aida`), identify which plots are available in them, and combine the MC and reference plots appropriately into a set of plot data files ending with `.dat`. More options are described by running `compare-histos --help`.

  Incidentally, the reference files for each Rivet analysis are to be found in the installed Rivet shared data directory, ⟨*installdir*⟩`/share/Rivet`. You can find the location of this by using the `rivet-config` command:
  `$ rivet-config --datadir`

- You can plot the created data files using the `make-plots` command:
  `$ make-plots --pdf *.dat`
  The `--pdf` flag makes the output plots in PDF format: by default the output is in PostScript (`.ps`), and flags for conversion to EPS and PNG are also available.

Part II
Selected analyses

Each Rivet release is accompanied by a standard library of analyses implementing currently a total of 250 experimental measurements or Monte-Carlo validation studies. The full listing of these is beyond the scope of this publication, but it is available both online at `http://rivet.hepforge.org/analyses` and as a part of the manual coming with each release of Rivet in the `doc/` sub-directory. Here, we only want to show-case a selection of analyses spanning the full spectrum of experiments from LEP over HERA to Tevatron and the LHC and demonstrating the versatility of the Rivet framework.

For each of the 250 analyses, in addition to a brief summary one can find information about the collider at which the measurement was made, references to the original publications, status and authors of the Rivet implementation as well as run details necessary for comparing a Monte-Carlo prediction with the data.

5. Selection of analyses available in the Rivet framework

5.1. ALEPH_1996_S3196992 [12]: Measurement of the quark to photon fragmentation function

Earlier measurements at LEP of isolated hard photons in hadronic Z decays, attributed to radiation from primary quark pairs, have been extended in the ALEPH experiment to include hard photon production inside hadron jets. Events are selected where all particles combine democratically to form hadron jets, one of which contains a photon with a fractional energy $z > 0.7$. After statistical subtraction of non-prompt photons, the quark-to-photon fragmentation function, $D(z)$, is extracted directly from the measured 2-jet rate.
BEAMS: $e^+ e^-$
ENERGIES: (45.6, 45.6) GeV
EXPERIMENT: ALEPH (LEP Run 1)
SPIRES ID: 3196992
STATUS: VALIDATED
AUTHORS:

- Frank Siegert ⟨ frank.siegert@cern.ch ⟩

Run details:

- $e^+e^- \to$ jets with π and η decays turned off.

5.2. ALICE_2011_S8945144 [13]:
Tranverse momentum spectra of pions, kaons and protons in pp collisions at 0.9 TeV

Obtaining the tranverse momentum spectra of pions, kaons and protons in pp collisions at $\sqrt{s} = 0.9$ TeV with ALICE at the LHC. Mean transverse momentum as a function of the mass of the emitted particle is also included.
Beams: $p\,p$
Energies: (450.0, 450.0) GeV
Experiment: ALICE (LHC)
Spires ID: 8945144
Status: VALIDATED
Authors:

- Pablo Bueno Gomez ⟨ UO189399@uniovi.es ⟩
- Eva Sicking ⟨ esicking@cern.ch ⟩

Run details:

- Diffractive events need to be enabled.

5.3. ARGUS_1993_S2653028 [14]:
Inclusive production of charged pions, kaons and protons in $\Upsilon(4S)$ decays.

Measurement of inclusive production of charged pions, kaons and protons from $\Upsilon(4S)$ decays. Kaon spectra are determined in two different ways using particle identification and detecting decays in-flight. Results are background continuum subtracted. This analysis is useful for tuning B meson decay modes.
Beams: $e^+\,e^-$
Energies: (5.3, 5.3) GeV
Spires ID: 2653028
Status: VALIDATED
Authors:

- Peter Richardson ⟨ Peter.Richardson@durham.ac.uk ⟩

Run details:

- e^+e^- analysis on the $\Upsilon(4S)$ resonance.

5.4. ATLAS_2012_I1094568 [15]:

Measurement of ttbar production with a veto on additional central jet activity

A measurement of the additional jet activity in dileptonic ttbar events. The fraction of events passing a veto requirement are shown as a function the veto scale for four central rapidity intervals. Two veto definitions are used: events are vetoed if they contain an additional jet in the rapidity interval with transverse momentum above a threshold, or alternatively, if the scalar transverse momentum sum of all additional jets in the rapidity interval is above a threshold.

BEAMS: $p\,p$
ENERGIES: (3500.0, 3500.0) GeV
EXPERIMENT: ATLAS (LHC)
SPIRES ID: 1094568
STATUS: VALIDATED
AUTHORS:

- Kiran Joshi ⟨ kiran.joshi@cern.ch ⟩

RUN DETAILS:

- Require dileptonic ttbar events at 7TeV.

5.5. BABAR_2007_S7266081 [16]:

Measurements of Semi-Leptonic Tau Decays into Three Charged Hadrons

Measurement of tau decays to three charged hadrons using a data sample corresponding to an integrated luminosity of 342 fb^{-1} collected with the BABAR detector at the SLAC PEP-II electron-positron storage ring operating at a center-of-mass energy near 10.58 GeV.

BEAMS: $e^+\,e^-$
ENERGIES: (3.5, 8.0) GeV
SPIRES ID: 7266081
STATUS: VALIDATED
AUTHORS:

- Peter Richardson ⟨ Peter.Richardson@durham.ac.uk ⟩

RUN DETAILS:

- Tau production, can be any process but original data was in e^+e^- at the $\Upsilon(4S)$ resonance, with CoM boost – 8.0 GeV (e) and 3.5 GeV (e^+)

5.6. BELLE_2006_S6265367 [17]:

Charm hadrons from fragmentation and B decays on the $\Upsilon(4S)$

Analysis of charm quark fragmentation at 10.6 GeV, based on a data sample of 103 fb collected by the Belle detector at the KEKB accelerator. Fragmentation into charm is studied for the main charmed hadron ground states, namely D^0, D^+, D_s^+ and Λ_c^+, as well as the excited states D^{*0} and D^{*+}. This analysis can be used to constrain charm fragmentation in Monte Carlo generators. As the original data are not corrected for the branching ratios of the decay modes used to observed the charm hadrons we also include distributions with unit normalisation which are more useful for Monte Carlo tuning.

BEAMS: $e^+\,e^-$

ENERGIES: (3.5, 8.0), (3.5, 7.9) GeV

SPIRES ID: 6265367

STATUS: VALIDATED

AUTHORS:

- Jan Eike von Seggern ⟨jan.eike.von.seggern@physik.hu-berlin.de⟩

RUN DETAILS:

- e^+e^- analysis on the $\Upsilon(4S)$ resonance, with CoM boost – 8.0 GeV (e) and 3.5 GeV (e^+)

5.7. CDF_2001_S4751469 [18]:

Field & Stuart Run I underlying event analysis.

The original CDF underlying event analysis, based on decomposing each event into a transverse structure with "toward", "away" and "transverse" regions defined relative to the azimuthal direction of the leading jet in the event. Since the toward region is by definition dominated by the hard process, as is the away region by momentum balance in the matrix element, the transverse region is most sensitive to multi-parton interactions. The transverse regions occupy $|\phi| \in [60°, 120°]$ for $|\eta| < 1$. The $p_\perp$ ranges for the leading jet are divided experimentally into the 'min-bias' sample from 0–20 GeV, and the 'JET20' sample from 18–49 GeV.

BEAMS: $\bar{p}\,p$

ENERGIES: (900.0, 900.0) GeV

EXPERIMENT: CDF (Tevatron Run 1)

SPIRES ID: 4751469

STATUS: VALIDATED

AUTHORS:

- Andy Buckley ⟨ andy.buckley@cern.ch ⟩

RUN DETAILS:

- $p\bar{p}$ QCD interactions at 1800 GeV. The leading jet is binned from 0–49 GeV, and histos can usually can be filled with a single generator run without kinematic sub-samples.

5.8. CLEO_2004_S5809304 [19]: Charm hadrons from fragmentation near the $\Upsilon(4S)$

Analysis of charm quark fragmentation at 10.5 GeV, based on a data sample of 103 fb collected by the CLEO experiment. Fragmentation into charm is studied for the charmed hadron ground states, namely D^0, D^+, as well as the excited states D^{*0} and D^{*+}. This analysis can be used to constrain charm fragmentation in Monte Carlo generators.
BEAMS: $e^+\,e^-$
ENERGIES: (5.3, 5.3) GeV
SPIRES ID: 6265367
STATUS: VALIDATED
AUTHORS:

- Peter Richardson ⟨ Peter.Richardson@durham.ac.uk ⟩

RUN DETAILS:

- e^+e^- analysis near the $\Upsilon(4S)$ resonance

5.9. CMS_2011_S8957746 [20]: Event shapes

Central transverse Thrust and Minor have been measured in proton-proton collisions at $\sqrt{s}$=7 TeV, with a data sample collected with the CMS detector at the LHC. The sample corresponds to an integrated luminosity of 3.2 inverse picobarns. Input for the variables are anti-kt jets with $R = 0.5$.
BEAMS: $p\,p$
ENERGIES: (3500.0, 3500.0) GeV
EXPERIMENT: CMS (LHC)
SPIRES ID: 8957746
STATUS: VALIDATED
AUTHORS:

- Hendrik Hoeth ⟨ hendrik.hoeth@cern.ch ⟩

Run details:

- pp QCD interactions at 7000 GeV. Particles with c*tau¿10mm are stable.

5.10. D0_2008_S7719523 [21]:
Isolated γ + jet cross-sections, differential in $p_\perp$ (γ) for various y bins

The process $p\bar{p} \to$ photon + jet + X as studied by the D0 detector at the Fermilab Tevatron collider at center-of-mass energy $\sqrt{s} = 1.96$ TeV. Photons are reconstructed in the central rapidity region $|y_\gamma| < 1.0$ with transverse momenta in the range 30–400 GeV, while jets are reconstructed in either the central $|y_{\text{jet}}| < 0.8$ or forward $1.5 < |y_{\text{jet}}| < 2.5$ rapidity intervals with $p_\perp^{\text{jet}} > 15$ GeV. The differential cross section $\mathrm{d}^3\sigma/\mathrm{d}p_\perp^\gamma \mathrm{d}y_\gamma \mathrm{d}y_{\text{jet}}$ is measured as a function of $p_\perp^\gamma$ in four regions, differing by the relative orientations of the photon and the jet. MC predictions have trouble with simultaneously describing the measured normalization and $p_\perp^\gamma$ dependence of the cross section in any of the four measured regions.
Beams: $\bar{p}\,p$
Energies: (980.0, 980.0) GeV
Experiment: D0 (Tevatron Run 2)
Spires ID: 7719523
Status: VALIDATED
Authors:

- Andy Buckley ⟨ andy.buckley@cern.ch ⟩
- Gavin Hesketh ⟨ gavin.hesketh@cern.ch ⟩
- Frank Siegert ⟨ frank.siegert@cern.ch ⟩

Run details:

- Produce only gamma + jet ($q, \bar{q}, g$) hard processes (for Pythia 6, this means MSEL=10 and MSUB indices 14, 29 & 115 enabled). The lowest bin edge is at 30 GeV, so a kinematic $p_\perp^{\text{min}}$ cut is probably required to fill the histograms.

5.11. DELPHI_1996_S3430090 [10]:
Delphi MC tuning on event shapes and identified particles.

Event shape and charged particle inclusive distributions measured using 750000 decays of Z bosons to hadrons from the DELPHI detector at LEP. This data, combined with identified particle distributions from all LEP experiments, was used for tuning of shower-hadronisation event generators by the original PROFESSOR method. This is a critical analysis for MC event generator tuning of final state radiation and both flavour and kinematic aspects of hadronisation models.
BEAMS: $e^+ e^-$
ENERGIES: (45.6, 45.6) GeV
EXPERIMENT: DELPHI (LEP 1)
SPIRES ID: 3430090
STATUS: VALIDATED
AUTHORS:

- Andy Buckley ⟨ andy.buckley@cern.ch ⟩
- Hendrik Hoeth ⟨ hendrik.hoeth@cern.ch ⟩

RUN DETAILS:

- $\sqrt{s} = 91.2$ GeV, $e^+e^- \to Z^0$ production with hadronic decays only

5.12. H1_2000_S4129130 [22]:
H1 energy flow in DIS

Measurements of transverse energy flow for neutral current deep- inelastic scattering events produced in positron-proton collisions at HERA. The kinematic range covers squared momentum transfers Q^2 from 3.2 to 2200 GeV^2; the Bjorken scaling variable x from 8×10^{-5} to 0.11 and the hadronic mass W from 66 to 233 GeV. The transverse energy flow is measured in the hadronic centre of mass frame and is studied as a function of Q^2, x, W and pseudorapidity. The behaviour of the mean transverse energy in the central pseudorapidity region and an interval corresponding to the photon fragmentation region are analysed as a function of Q^2 and W. This analysis is useful for exploring the effect of photon PDFs and for tuning models of parton evolution and treatment of fragmentation and the proton remnant in DIS.

BEAMS: $p\,e^+$
ENERGIES: (820.0, 27.5) GeV
EXPERIMENT: H1 (HERA)
SPIRES ID: 4129130
STATUS: VALIDATED
AUTHORS:

- Peter Richardson ⟨ peter.richardson@durham.ac.uk ⟩

RUN DETAILS:

- e^+p deep inelastic scattering with p at 820 GeV, e^+ at 27.5 GeV $\rightarrow \sqrt{s}$ = 300 GeV

5.13. JADE_1998_S3612880 [23]:
Event shapes for 22, 35 and 44 GeV

Thrust, Jet Mass and Broadenings, Y23 for 35 and 44 GeV and only Y23 at 22 GeV.
BEAMS: $e^-\,e^+$
ENERGIES: (11.0, 11.0), (17.5, 17.5), (22.0, 22.0) GeV
EXPERIMENT: JADE (PETRA)
SPIRES ID: 3612880
STATUS: VALIDATED
AUTHORS:

- Holger Schulz ⟨ holger.schulz@physik.hu-berlin.de ⟩

RUN DETAILS:

- Z→hadronic final states, bbar contributions have been corrected for as well as ISR

5.14. LHCB_2011_I919315 [24]:
Inclusive differential Φ production cross-section as a function of p_T and y

Measurement of the inclusive differential Φ cross-section in pp collisions at $\sqrt{s} = 7$TeV in the rapidity range of $2.44 < y < 4.06$ and the $p_{\rm T}$ range of 0.6 GeV/c $< p_{\rm T} <$ 5.0 GeV/c.
BEAMS: $p\,p$
ENERGIES: (3500.0, 3500.0) GeV

EXPERIMENT: LHCB (LHC)
STATUS: VALIDATED
AUTHORS:

- Friederike Blatt ⟨ friederike.blatt@tu-dortmund.de ⟩
- Michael Kaballo ⟨ michael.kaballo@tu-dortmund.de ⟩
- Till Moritz Karbach ⟨ moritz.karbach@tu-dortmund.de ⟩

RUN DETAILS:

- pp collisions, QCD-Events, $\sqrt{s} = 7$TeV

5.15. LHCF_2012_I1115479 [25]:

Measurement of forward neutral pion transverse momentum spectra for $\sqrt{s} = 7$ TeV proton-proton collisions at LHC

The inclusive production rate of neutral pions has been measured by LHCf experiment during $\sqrt{s} = 7$ TeV pp collision operation in early 2010. In order to ensure good event reconstruction efficiency, the range of the π^0 rapidity and $p_\perp$ are limited to $8.9 < y < 11.0$ and $p_\perp < 0.6$ GeV, respectively.
BEAMS: $p\,p$
ENERGIES: (3500.0, 3500.0) GeV
EXPERIMENT: LHCF (LHC)
STATUS: VALIDATED
AUTHORS:

- Sercan Sen ⟨ ssen@cern.ch ⟩

RUN DETAILS:

- Inelastic events (ND+SD+DD) at $\sqrt{s} = 7$ TeV.

5.16. OPAL_2004_S6132243 [26]:

Event shape distributions and moments in $e^+e^- \to$ hadrons at 91–209 GeV

Measurement of e^+e^- event shape variable distributions and their 1st to 5th moments in LEP running from the Z pole to the highest LEP 2 energy of 209 GeV.
BEAMS: $e^+\,e^-$
ENERGIES: (45.6, 45.6), (66.5, 66.5), (88.5, 88.5), (98.5, 98.5) GeV

EXPERIMENT: OPAL (LEP 1 & 2)
SPIRES ID: 6132243
STATUS: VALIDATED
AUTHORS:

- Andy Buckley ⟨ andy.buckley@cern.ch ⟩

RUN DETAILS:

- Hadronic e^+e^- events at 4 representative energies (91, 133, 177, 197). Runs need to have ISR suppressed, since the analysis was done using a cut of $\sqrt{s} - \sqrt{s_{\text{reco}}} < 1\,\text{GeV}$. Particles with a livetime $> 3 \cdot 10^{-10}\,\text{s}$ are considered to be stable.

5.17. PDG_HADRON_MULTIPLICITIES [27]: Hadron multiplicities in hadronic e^+e^- events

Hadron multiplicities in hadronic e^+e^- events, taken from Review of Particle Properties 2008, table 40.1, page 355. Average hadron multiplicities per hadronic e^+e^- annihilation event at $\sqrt{s} \approx$ 10, 29–35, 91, and 130–200 GeV. The numbers are averages from various experiments. Correlations of the systematic uncertainties were considered for the calculation of the averages.
BEAMS: $e^+\,e^-$
ENERGIES: (5.0, 5.0), (17.5, 17.5), (45.6, 45.6), (88.5, 88.5) GeV
EXPERIMENT: PDG (Various)
SPIRES ID: 7857373
STATUS: VALIDATED
AUTHORS:

- Hendrik Hoeth ⟨ hendrik.hoeth@cern.ch ⟩

RUN DETAILS:

- Hadronic events in e^+e^- collisions

5.18. SLD_2004_S5693039 [28]: Production of π^+, π^-, K^+, K^-, p and $\bar{p}$ in Light (uds), c and b Jets from Z Decays

Measurements of the differential production rates of stable charged particles in hadronic Z^0 decays, and of charged pions, kaons and protons identified over a wide momentum range. In addition to flavour-inclusive Z^0 decays,

measurements are made for Z^0 decays into light (u, d, s), c and b primary flavors.
BEAMS: $e^+ e^-$
ENERGIES: (45.6, 45.6) GeV
EXPERIMENT: SLD (SLC)
SPIRES ID: 5693039
STATUS: VALIDATED
AUTHORS:

- Peter Richardson ⟨ Peter.Richardson@durham.ac.uk ⟩

RUN DETAILS:

- Hadronic Z decay events generated on the Z pole ($\sqrt{s} = 91.2$ GeV)

5.19. STAR_2006_S6500200 [29]: Identified hadron spectra in pp at 200 GeV

$p_\perp$ distributions of charged pions and (anti)protons in pp collisions at $\sqrt{s} = 200$ GeV, measured by the STAR experiment at RHIC in non-single-diffractive minbias events.
BEAMS: $p\,p$
ENERGIES: (100.0, 100.0) GeV
EXPERIMENT: STAR (RHIC pp 200 GeV)
SPIRES ID: 6500200
STATUS: VALIDATED
AUTHORS:

- Bedanga Mohanty ⟨ bedanga@rcf.bnl.gov ⟩
- Hendrik Hoeth ⟨ hendrik.hoeth@cern.ch ⟩

RUN DETAILS:

- pp at 200 GeV

5.20. TASSO_1990_S2148048 [30]: Event shapes in e+e- annihilation at 14-44 GeV

Event shapes Thrust, Sphericity, Aplanarity at four different energies
BEAMS: $e^- e^+$
ENERGIES: (7.0, 7.0), (11.0, 11.0), (17.5, 17.5), (21.9, 21.9) GeV

Experiment: TASSO (PETRA)
Spires ID: 2148048
Status: VALIDATED
Authors:

- Holger Schulz ⟨ holger.schulz@physik.hu-berlin.de ⟩

Run details:

- $e^+e^- \to$ jet jet (+ jets) Kinematic cuts such as CKIN(1) in Pythia need to be set slightly below the CMS energy.

5.21. TOTEM_2012_I1115294 [31]: Forward dN/deta at 7 TeV

The TOTEM experiment has measured the charged particle pseudorapidity density $dN_{\mathrm{ch}}/d\eta$ in pp collisions at $\sqrt{s} = 7\,\mathrm{TeV}$ for $5.3 < |\eta| < 6.4$ in events with at least one charged particle with transverse momentum above 40 MeV/c in this pseudorapidity range.
Beams: $p\,p$
Energies: (3500.0, 3500.0) GeV
Experiment: TOTEM (LHC)
Status: VALIDATED
Authors:

- Hendrik Hoeth ⟨ hendrik.hoeth@cern.ch ⟩

Run details:

- pp QCD interactions at 900 GeV and 7 TeV.

5.22. UA1_1990_S2044935 [32]: UA1 multiplicities, transverse momenta and transverse energy distributions.

Particle multiplicities, transverse momenta and transverse energy distributions at the UA1 experiment, at energies of 200, 500 and 900 GeV (with one plot at 63 GeV for comparison).
Beams: $\bar{p}\,p$
Energies: (31.5, 31.5), (100.0, 100.0), (250.0, 250.0), (450.0, 450.0) GeV
Experiment: UA1 (SPS)
Spires ID: 2044935
Status: VALIDATED
Authors:

- Andy Buckley ⟨ andy.buckley@cern.ch ⟩
- Christophe Vaillant ⟨ c.l.j.j.vaillant@durham.ac.uk ⟩

Run details:

- QCD min bias events at sqrtS = 63, 200, 500 and 900 GeV.

5.23. UA5_1982_S875503 [33]:
UA5 multiplicity and pseudorapidity distributions for pp and $p\bar{p}$.

Comparisons of multiplicity and pseudorapidity distributions for pp and $p\bar{p}$ collisions at 53 GeV, based on the UA5 53 GeV runs in 1982. Data confirms the lack of significant difference between the two beams.
Beams: $\bar{p}\,p$, $p\,p$
Energies: (26.5, 26.5) GeV
Experiment: UA5 (SPS)
Spires ID: 875503
Status: VALIDATED
Authors:

- Andy Buckley ⟨ andy.buckley@cern.ch ⟩
- Christophe Vaillant ⟨ c.l.j.j.vaillant@durham.ac.uk ⟩

Run details:

- Min bias QCD events at $\sqrt{s} = 53$ GeV. Run with both pp and $p\bar{p}$ beams.

Part III
How Rivet works

Hopefully by now you've run Rivet a few times and got the hang of the command line interface and viewing the resulting analysis data files. Maybe you've got some ideas of analyses that you would like to see in Rivet's library. If so, then you'll need to know a little about Rivet's internal workings before you can start coding: with any luck by the end of this section that won't seem particularly intimidating.

The core objects in Rivet are "projections" and "analyses". Hopefully "analyses" isn't a surprise — that's just the collection of routines that will make histograms to compare with reference data, and the only things that might differ there from experiences with HZTool[34] are the new histogramming system and the fact that we've used some object orientation concepts to make life a bit easier. The meaning of "projections", as applied to event analysis, will probably be less obvious. We'll discuss them soon, but first a semi-philosophical aside on the "right way" to do physics analyses on and involving simulated data.

6. The science and art of physically valid MC analysis

The world of MC event generators is a wonderfully convenient one for experimentalists: we are provided with fully exclusive events whose most complex correlations can be explored and used to optimise analysis algorithms and some kinds of detector correction effects. It is absolutely true that the majority of data analyses and detector designs in modern collider physics would be very different without MC simulation.

But it is very important to remember that it is just simulation: event generators encode much of known physics and phenomenologically explore the non-perturbative areas of QCD, but only unadulterated experiment can really tell us about how the world behaves. The richness and convenience of MC simulation can be seductive, and it is important that experimental use of MC strives to understand and minimise systematic biases which may result from use of simulated data, and to not "unfold" imperfect models when measuring the real world. The canonical example of the latter effect is the unfolding of hadronisation (a deeply non-perturbative and imperfectly-understood process) at the Tevatron (Run I), based on MC models. Publishing "measured quarks"

is not physics — much of the data thus published has proven of little use to either theory or experiment in the following years. In the future we must be alert to such temptation and avoid such gaffes — and much more subtle ones.

These concerns on how MC can be abused in treating measured data also apply to MC validation studies. A key observable in QCD tunings is the $p_\perp$ of the Z boson, which has no phase space at exactly $p_\perp = 0$ but a very sharp peak at $\mathcal{O}$(1-2 GeV). The exact location of this peak is mostly sensitive to the width parameter of a nucleon "intrinsic $p_\perp$" in MC generators, plus some soft initial state radiation and QED bremstrahlung. Unfortunately, all the published Tevatron measurements of this observable have either "unfolded" the QED effects to the "Z $p_\perp$" as attached to the object in the HepMC/HEPEVT event record with a PDG ID code of 23, or have used MC data to fill regions of phase space where the detector could not measure. Accordingly, it is very hard to make an accurate and portable MC analysis to fit this data, without similarly delving into the event record in search of "the boson". While common practice, this approach intrinsically limits the precision of measured data to the calculational order of the generator — often not analytically well-defined. We can do better.

Away from this philosophical propaganda (which nevertheless we hope strikes some chords in influential places...), there are also excellent pragmatic reasons for MC analyses to avoid treating the MC "truth" record as genuine truth. The key argument is portability: there is no MC generator which is the ideal choice for all scenarios, and an essential tool for understanding sub-leading variability in theoretical approaches to various areas of physics is to use several generators with similar leading accuracies but different sub-leading formalisms. While the HEPEVT record as written by HERWIG and PYTHIA has become familiar to many, there are many ambiguities in how it is filled, from the allowed graph structures to the particle content. Notably, the Sherpa event generator explicitly elides Feynman diagram propagators from the event record, perhaps driven by a desire to protect us from our baser analytical instincts. The Herwig++ event generator takes the almost antipodal approach of expressing different contributing Feynman diagram topologies in different ways (*not* physically meaningful!) and seamlessly integrating shower emissions with the hard process particles. The general trend in MC simulation is to blur the practically-induced line between the sampled matrix element and the Markovian parton cascade, challenging many established assumptions about "how MC works". In short, if you want to "find" the Z to see what its $p_\perp$ or η spectrum looks like, many new generators

may break your honed PYTHIA code... or silently give systematically wrong results. The unfortunate truth is that most of the event record is intended for generator debugging rather than physics interpretation.

Fortunately, the situation is not altogether negative: in practice it is usually as easy to write a highly functional MC analysis using only final state particles and their physically meaningful on-shell decay parents. These are, since the release of HepMC 2.5, standardised to have status codes of 1 and 2 respectively. Z-finding is then a matter of choosing decay lepton candidates, windowing their invariant mass around the known Z mass, and choosing the best Z candidate: effectively a simplified version of an experimental analysis of the same quantity. This is a generally good heuristic for a safe MC analysis! Note that since it's known that you will be running the analysis on signal events, and there are no detector effects to deal with, almost all the details that make a real analysis hard can be ignored. The one detail that is worth including is summing momentum from photons around the charged leptons, before mass-windowing: this physically corresponds to the indistinguishability of collinear energy deposits in trackers and calorimeters and would be the ideal published experimental measurement of Drell-Yan $p_\perp$ for MC tuning. Note that similar analyses for W bosons have the luxury over a true experiment of being able to exactly identify the decay neutrino rather than having to mess around with missing energy. Similarly, detailed unstable hadron (or tau) reconstruction is unnecessary, due to the presence of these particles in the event record with status code 2. In short, writing an effective analysis which is automatically portable between generators is no harder than trying to decipher the variable structures and multiple particle copies of the debugging-level event objects. And of course Rivet provides lots of tools to do almost all the standard fiddly bits for you, so there's no excuse!

Good luck, and be careful!

7. Projections

The name "projection" is meant to evoke thoughts of projection operators, low-dimensional slices/views of high-dimensional spaces, and other things that might appeal to physicists who view the world through quantum-tinted lenses. A more mundane, but equally applicable, name would be "observable calculators", but since that's a long name, the things they return aren't *necessarily* observable, and they all inherit from the `Projection` base class,

we'll stick to that name. It doesn't take long to get used to using the name as a synonym for "calculator", without being intimidated by ideas that they might be some sort of high-powered deep magic. 90% of them is simple and self-explanatory, as a peek under the bonnet of e.g. the all-important `FinalState` projection will reveal.

Projections can be relatively simple things like event shapes (i.e. scalar, vector or tensor quantities), or arbitrarily complex things like lossy or selective views of the event final state. Most users will see them attached to analyses by declarations in each analysis' initialisation, but they can also be recursively "nested" inside other projections[2] (provided there are no infinite loops in the nesting chain.) Calling a complex projection in an analysis may actually transparently execute many projections on each event.

7.1. Projection caching

Aside from semantic issues of how the class design assigns the process of analysing events, projections are important computationally because they live in a framework which automatically stores ("caches") their results between events. This is a crucial feature for the long-term scalability of Rivet, as the previous experience with HZTool was that HERA validation code ran very slowly due to repeated calculation of the same $k_\perp$ clustering algorithm (at that time notorious for scaling as the 3rd power of the number of particles.)

A concrete example may help in understanding how this works. Let's say we have two analyses which have the same run conditions, i.e. incoming beam types, beam energies, etc. Each also uses the thrust event shape measure to define a set of basis vectors for their analysis. For each event that gets passed to Rivet, whichever analysis gets called first will immediately (although maybe indirectly) call a `FinalState` projection to get a list of stable, physical particles (filtering out the intermediate and book-keeping entries in the HepMC event record). That FS projection is then "attached" to the event. Next, the first analysis will call a `Thrust` projection which internally uses the same final state projection to define the momentum vectors used in calculating the thrust. Once finished, the thrust projection will also be attached to the event.

[2]Provided there are no dependency loops in the projection chains! Strictly, only acyclic graphs of projection dependencies are valid, but there is currently no code in Rivet that will attempt to verify this restriction.

So far, projections have offered no benefits. However, when the second analysis runs it will similarly try to apply its final state and thrust projections to the event. Rather than repeat the calculations, Rivet's infrastructure will detect that an equivalent calculation has already been run and will just return references to the already-run projections. Since projections can also contain and use other projections, this model allows some substantial computational savings, without the analysis author even needing to be particularly aware of what is going on.

Observant readers may have noticed a problem with all this projection caching cleverness: what if the final states aren't defined the same way? One might provide charged final state particles only, or the acceptances (defined in pseudorapidity range and a IR $p_\perp$ cutoff) might differ. Rivet handles this by making each projection provide a comparison operator which is used to decide whether the cached version is acceptable or if the calculation must be re-run with different settings. Because projections can be nested, applying a top-level projection to an event can spark off a cascade of comparisons, calculations and cache accesses, making use of existing results wherever possible.

7.2. Using projection caching

So far this is all theory — how does one actually use projections in Rivet? First, you should understand that projections, while semantically stored within each other, are actually all registered with a central `ProjectionHandler` object.[3] The reason for this central registration is to ensure that all projections' lifespans are managed in a consistent way, and to protect projection and analysis authors from some technical subtleties in how C++ polymorphism works.

Inside the constructor of a `Projection` or the `init` method of an `Analysis` class, you must call the `addProjection` function. This takes two arguments, the projection to be registered (by `const` reference), and a name. The name is local to the parent object, so you need not worry about name clashes between objects. A very important point is that the passed `Projection` is not the one that is actually centrally registered — that distinction belongs to a newly created heap object which is created within the `addProjection` method by means of the overloaded `Projection::clone()` method. Hence

[3] As of version 1.1 onwards — previously, they were stored as class members inside other `Projection` s and `Analysis` classes.

it is completely safe — and recommended — to use only local (stack) objects in `Projection` and `Analysis` constructors.

At this point, if you have rightly bought into C++ ideas like super-strong type-safety, this proliferation of dynamic casting may worry you: the compiler can't possibly check if a projection of the requested name has been registered, nor whether the downcast to the requested concrete type is legal. These are very legitimate concerns!
In truth, we'd like to have this level of extra safety! But in the past, when projections were held as members of `ProjectionApplier` *classes rather than in the central* `ProjectionHandler` *repository, the benefits of the strong typing were outweighed by more serious and subtle bugs relating to projection lifetime and object "slicing". At least when the current approach goes wrong it will throw an unmissable* runtime *error — until it's fixed, of course! — rather than silently do the wrong thing.*
Our problems here are a microcosm of the perpetual language battle between strict and dynamic typing, runtime versus compile time errors. In practice, this manifests itself as a trade-off between the benefits of static type safety and the inconvenience of the type-system gymnastics that it engenders. We take some comfort from the number of very good programs have been and are still written in dynamically typed, interpreted languages like Python, where virtually all error checking (barring first-scan parsing errors) must be done at runtime. By pushing some *checking to the domain of runtime errors, Rivet's code is (we believe) in practice safer, and certainly more clear and elegant. However, we believe that with runtime checking should come a culture of unit testing, which is not yet in place in Rivet.*
As a final thought, one reason for Rivet's internal complexity is that C++ is just not a very good language for this sort of thing: we are operating on the boundary between event generator codes, number crunching routines (including third party libraries like FastJet) and user routines. The former set unavoidably require native interfaces and benefit from static typing; the latter benefit from interface flexibility, fast prototyping and syntactic clarity. Maybe a future version of Rivet will break through the technical barriers to a hybrid approach and allow users to run compiled projections from interpreted analysis code. For now, however, we hope that our brand of "slightly less safe C++" will be a pleasant compromise.

---

8. Analyses

8.1. Writing a new analysis

This section provides a recipe that can be followed to write a new analysis using the Rivet projections.

Every analysis must inherit from `Rivet::Analysis` and, in addition to the constructor, must implement a minimum of three methods. Those methods are `init()`, `analyze(const Rivet::Event&)` and `finalize()`, which are called once at the beginning of the analysis, once per event and once at the end of the analysis respectively.

The new analysis should include the header for the base analysis class plus whichever Rivet projections are to be used, and should work under the `Rivet` namespace. Since analyses are hardly ever intended to be inherited from, they are usually implemented within a single `.cc` file with no corresponding header. The skeleton of a new analysis named `UserAnalysis` that uses the `FinalState` projection might therefore start off looking like this, in a file named `UserAnalysis.cc`:

```
#include "Rivet/Analysis.hh"

namespace Rivet {

  class UserAnalysis : public Analysis {
  public:
    UserAnalysis() : Analysis("USERANA") { }
    void init() { ... }
    void analyze(const Event& event) { ... }
    void finalize() { ... }
  };

}
```

The constructor body is usually left empty, as all event loop setup is done in the `init()` method: the one *required* constructor feature is to make a call to its base `Analysis` constructor, passing a string by which the analysis will *register* itself with the Rivet framework. This name is the one exposed to a command-line or API user of this analysis: usually it is the same as the class name, which for official analyses is always in upper case.

Early versions of Rivet required the user to declare allowed beam types, energies, whether a cross-section is required, etc. in the analysis constructor via methods like `setBeams(...)` and `setNeedsCrossSection(...)`. This information is now *much* preferred to be taken from the `.info` file for the analysis, and *must* be done this way in analyses submitted for inclusion in future Rivet releases.

The `init()` method for the `UserAnalysis` class should add to the analysis all of the projections that will be used. Projections can be added to an analysis with a call to `addProjection(Projection, std::string)`, which takes as argument the projection to be added and a name by which that projection can later be referenced. For this example the `FinalState` projection is to be referenced by the string `"FS"` to provide access to all of the final state particles inside a detector pseudorapidity coverage of ± 5.0. The syntax to create and add that projection is as follows:

```
init() {
  const FinalState fs(-5.0, 5.0);
  addProjection(fs, "FS");
}
```

A second task of the `init()` method is the booking of all histograms which are later to be filled in the analysis code. Information about the histogramming system can be found in Section 8.3.

8.2. Utility classes

Rivet provides quite a few object types for physics purposes, such as three- and four-vectors, matrices and Lorentz boosts, and convenience proxy objects for e.g. particles and jets. We now briefly summarise the most important features of some of these objects; more complete interface descriptions can be found in the generated Doxygen web pages on the Rivet web site, or simply by browsing the relevant header files.

8.2.1. `FourMomentum`

The `FourMomentum` class is the main physics vector that you will encounter when writing Rivet analyses. Its functionality and interface are similar to the CLHEP `HepLorentzVector` with which many users will be familiar, but without some of the historical baggage.

Vector components. The `FourMomentum` `E()`, `px()`, `py()`, `pz()` & `mass()` methods are (unsurprisingly) accessors for the vector's energy, momentum components and mass. The `vector3()` method returns a spatial `Vector3` object, i.e. the 3 spatial components of the 4-vector.

Useful properties. The `pT()` and `Et()` methods are used to calculate the transverse momentum and transverse energy. Angular variables are accessed via the `eta()`, `phi()` and `theta()` for the pseudorapidity, azimuthal angle and polar angle respectively. More explicitly named versions of these also exist, named `pseudorapidity()`, `azimuthalAngle()` and `polarAngle()`. Finally, the true rapidity is accessed via the `rapidity()` method. Many of these functions are also available as external functions, as are algebraic functions such as `cross(vec3a, vec3b)`, which is perhaps more palatable than `vec3a.cross(vec3b)`.

Distances. The η–ϕ distance between any two four-vectors (and/or three-vectors) can be computed using a range of overloaded external functions of the type `deltaR(vec1, vec2)`. Angles between such vectors can be calculated via the similar `angle(vec1, vec2)` functions.

8.2.2. `Particle`

This class is a wrapper around the HepMC `GenParticle` class. `Particle` objects are usually obtained as a vector from the `particles()` method of a `FinalState` projection. Rather than having to directly use the HepMC objects, and e.g. translate HepMC four-vectors into the Rivet equivalent, several key properties are accessed directly via the `Particle` interface (and more may be added). The main methods of interest are `momentum()`, which returns a `FourMomentum`, and `pdgId()`, which returns the PDG particle ID code. The PDG code can be used to access particle properties by using functions such as `PID::isHadron()`, `PID::threeCharge()`, etc. (these are defined in `Rivet/Tools/ParticleIDMethods.hh`.)

8.2.3. `Jet`

Jets are obtained from one of the jet accessor methods of a projection that implements the `JetAlg` interface, e.g. `FastJets::jetsByPt()` (this returns the jets sorted by $p_\perp$, such that the first element in the vector is the hardest jet — usually what you want.) The most useful methods are `particles()`, `momenta()`, `momentum()` (a representative `FourMomentum`), and some checks

on the jet contents such as `containsParticleId(pid)`, `containsCharm()` and `containsBottom()`.

8.2.4. Mathematical utilities

The `Rivet/Math/MathUtils.hh` header defines a variety of mathematical utility functions. These include testing functions such as `isZero(a)`, `fuzzyEquals(a, b)` and `inRange(a, low, high)`, whose purpose is hopefully self-evident, and angular range-mapping functions such as `mapAngle0To2Pi(a)`, `mapAngleMPiToPi(a)`, etc.

8.3. Histogramming

Rivet's histogramming uses the AIDA interfaces, composed of abstract classes `IHistogram1D`, `IProfile1D`, `IDataPointSet` etc. which are built by a factories system. Since it's our feeling that much of the factory infrastructure constitutes an abstraction overload, we provide histogram booking functions as part of the `Analysis` class, so that in the `init` method of your analysis you should book histograms with function calls like:

```
void init() {
  _h_one = bookHistogram1D(2,1,1);
  _h_two = bookProfile1D(3,1,2);
  _h_three = bookHistogram1D("d00-x00-y00", 50, 0.0, 1.0);
}
```

Here the first two bookings have a rather cryptic 3-integer sequence as the first arguments. This is the recommended scheme, as it makes use of the exported data files from HepData, in which 1D histograms are constructed from a combination of x and y axes in a dataset d, corresponding to names of the form `d`$\langle d \rangle$`-x`$\langle x \rangle$`-y`$\langle y \rangle$. This auto-booking of histograms saves you from having to copy out reams of bin edges and values into your code, and makes sure that any data fixes in HepData are easily propagated to Rivet. The reference data files which are used for these booking methods are distributed and installed with Rivet, you can find them in the $\langle$*installdir*$\rangle$`/share/Rivet` directory of your installation. The third booking is for a histogram for which there is no such HepData entry: it uses the usual scheme of specifying the name, number of bins and the min/max x-axis limits manually.

Filling the histograms is done in the `MyAnalysis::analyse()` function. Remember to specify the event weight as you fill:

```
void analyze(const Event& e) {
  [projections, cuts, etc.]
  ...
  _h_one->fill(pT, event.weight());
  _h_two->fill(pT, Nch, event.weight());
  _h_three->fill(fabs(eta), event.weight());
}
```

Finally, histogram normalisations, scalings, divisions etc. are done in the `MyAnalysis::finalize()` method. For normalisations and scalings you will find appropriate convenience methods `Analysis::normalize(histo, norm)` and `Analysis::scale(histo, scalefactor)`. Many analyses need to be scaled to the generator cross-section, with the number of event weights to pass cuts being included in the normalisation factor: for this you will have to track the passed-cuts weight sum yourself via a member variable, but the analysis class provides `Analysis::crossSection()` and `Analysis::sumOfWeights()` methods to access the pre-cuts cross-section and weight sum respectively.

8.4. Analysis metadata

To keep the analysis source code uncluttered, and to allow for iteration of data plot presentation without re-compilation and/or re-running, Rivet prefers that analysis metadata is provided via separate files rather than hard-coded into the analysis library. There are two such files: an *analysis info* file, with the suffix `.info`, and a *plot styling* file, with the suffix `.plot`.

8.4.1. Analysis info files

The analysis info files are in YAML format: a simple data format intended to be cleaner and more human-readable/writeable than XML. As well as the analysis name (which must coincide with the filename and the name provided to the `Analysis` constructor, this file stores details of the collider, experiment, date of the analysis, Rivet/data analysis authors and contact email addresses, one-line and more complete descriptions of the analysis, advice on how to run it, suggested generator-level cuts, and BibTeX keys and entries for this user manual. It is also where the validation status of the analysis is declared:

See the standard analyses' info files for guidance on how to populate this file. Info files are searched for in the paths known to the `Rivet::getAnalysisInfoPaths()` function, which many be prepended to using the `$RIVET_INFO_PATH` environment variable: the first matching file to be found will be used.

8.4.2. Plot styling files

The `.plot` files are in the header format for the `make-plots` plotting system and are picked up and merged with the plot data by the Rivet `compare-histos` script which produces the `make-plots` input data files. All the analysis' plots should have a `BEGIN PLOT ... END PLOT` section in this file, specifying the title and x/y-axis labels (the `Title`, and `XLabel`/`YLabel` directives). In addition, you can use this file to choose whether the x and/or y axes should be shown with a log scale (`LogX`, `LogY`), to position the legend box to minimise clashes with the data points and MC lines (`LegendXPos`, `LegendYPos`) and any other valid `make-plots` directives including special text labels or forced plot range boundaries. Regular expressions may be used to apply a directive to all analysis names matching a pattern rather than having to specify the same directive repeatedly for many plots.

See the standard analyses' plot files and the `make-plots` documentation (e.g. on the Rivet website) for guidance on how to write these files. Plot info files are searched for in the paths known to the `Rivet::getAnalysisPlotPaths()` function, which many be prepended to using the `$RIVET_PLOT_PATH` environment variable. As usual, the first matching file to be found will be used.

8.5. Pluggable analyses

Rivet's standard analyses are not actually built into the main `libRivet` library: they are loaded dynamically at runtime as an analysis *plugin library*. While you don't need to worry too much about the technicalities of this, it does mean that you can similarly write analyses of your own, compile them into a similar plugin library and run them from `rivet` without ever having to modify any of the main Rivet sources or build system. This means that you can write and run your own analyses with a system-installed copy of Rivet, and not have to re-patch the main library when a newer version comes out (although chances are you will have to recompile, since the binary interface usually change between releases.)

To get started writing your analysis and understand the plugin system better, you should check out the documentation in the wiki on the Rivet website: `http://rivet.hepforge.org/trac/wiki/`. The standard `rivet-mkanalysis` and `rivet-buildplugin` scripts can respectively be used to make an analysis template with many "boilerplate" details filled in (including bibliographic information from Inspire if available), and to build a plugin library with the appropriate compiler options.

8.5.1. Plugin paths

To load pluggable analyses you will need to set the `$RIVET_ANALYSIS_PATH` environment variable: this is a standard colon-separated UNIX path, specifying directories in which analysis plugin libraries may be found. If it is unspecified, the Rivet loader system will assume that the only entry is the `lib` directory in the Rivet installation area. Specifying the variable adds new paths for searching *before* the standard library area, and they will be searched in the left-to-right order in the path variable. If analyses with duplicate names are found, a warning message is issued and the first version to have been found will be used. This allows you to override standard analyses with same-named variants of your own, provided they are loaded from different directories.

Several further environment variables are used to load analysis reference data and metadata files:

`$RIVET_REF_PATH`: A standard colon-separated path list, whose elements are searched in order for reference histogram files. If the required file is not found in this path, Rivet will fall back to looking in the analysis library paths (for convenience, as it is normal for plugin analysis developers to put analysis library and data files in the same directory and it would be annoying to have to set several variables to make this work), and then the standard Rivet installation data directory.

`$RIVET_INFO_PATH`: The path list searched first for analysis `.info` metadata files. The search fallback mechanism works as for `$RIVET_REF_PATH`.

`$RIVET_PLOT_PATH`: The path list searched first for analysis `.plot` presentation style files. The search fallbacks again work as for `$RIVET_REF_PATH`.

These paths can be accessed from the API using the `Rivet::getAnalysisLibPaths()` etc. functions, and can be searched for files using the Rivet lookup rules via the `Rivet::findAnalysisLibFile(filename)` etc. functions. These functions are also available in the Python `rivet` module. See the Doxygen documentation for more details.

9. Using Rivet as a library

You don't have to use Rivet via the provided command-line programmes: for some applications you may want to have more direct control of how Rivet processes events. Here are some possible reasons:

- You need to not waste CPU cycles and I/O resources on rendering HepMC events to a string representation which is immediately read back in. The FIFO idiom (Section 3.1) is not perfect: we use it in circumstances where the convenience and decoupling outweighs the CPU cost.
- You don't want to write out histograms to file, preferring to use them as code objects. Perhaps for applications which want to manipulate histogram data periodically before the end of the run.
- You enjoy tormenting Rivet developers who know their API is far from perfect, by complaining if it changes!
- ... and many more!

The Rivet API (application programming interface) has been designed in the hope of very simple integration into other applications: all you have to do is create a `Rivet::AnalysisHandler` object, tell it which analyses to apply on the events, and then call its `analyse(evt)` method for each HepMC event – wherever they come from. The API is (we hope) stable, with the exception of the histogramming parts.

The histogramming interfaces in Rivet have long been advertised as marked for replacement, and while progress in that area has lagger far behind our ambitions, it *will* happen with the 2.0.0 release, with unavoidable impact on the related parts of the API. You have been warned!

The API is available for C++ and, in a more restricted form, Python. We will explain the C++ version here; if you wish to operate Rivet (or e.g. use its path-searching capabilities to find Rivet-related files in the standard way) from Python then take a look inside the `rivet` and `rivet-*` Python scripts (e.g. `less ‘which rivet‘`) or use the module documentation cf.

```
> python
>>> import rivet
>>> help(rivet)
```

And now the C++ API. The best way to explain is, of course, by example. Here is a simple C++ example based on the `test/testApi.cc` source which we use in development to ensure continuing API functionality:

```
#include "Rivet/AnalysisHandler.hh"
#include "HepMC/GenEvent.h"
#include "HepMC/IO_GenEvent.h"

using namespace std;

int main() {

  // Create analysis handler
  Rivet::AnalysisHandler rivet;

  // Specify the analyses to be used
  rivet.addAnalysis("D0_2008_S7554427");
  vector<string> moreanalyses(1, "D0_2007_S7075677");
  rivet.addAnalyses(moreanalyses);

  // The usual mess of reading from a HepMC file!
  std::istream* file = new std::fstream("testApi.hepmc", std::ios::in);
  HepMC::IO_GenEvent hepmcio(*file);
  HepMC::GenEvent* evt = hepmcio.read_next_event();
  double sum_of_weights = 0.0;
  while (evt) {
    // Analyse the current event
    rivet.analyze(*evt);
    sum_of_weights += evt->weights()[0];

    // Clean up and get next event
    delete evt; evt = 0;
    hepmcio >> evt;
  }
  delete file; file = 0;

  rivet.setCrossSection(1.0);
  rivet.setSumOfWeights(sum_of_weights); // not necessary, but allowed
  rivet.finalize();
  rivet.writeData("out");

  return 0;
```

```
}
```

Compilation of this, if placed in a file called `myrivet.cc`, into an executable called `myrivet` is simplest and most robust with use of the `rivet-config` script:

```
g++ myrivet.cc -o myrivet `rivet-config --cppflags --ldflags --libs`
```

It *should* just work!

If you are doing something a bit more advanced, for example using the AGILe package's similar API to generate Fortran generator Pythia events and pass them directly to the Rivet analysis handler, you will need to also add the various compiler and linker flags for the extra libraries, e.g.

```
g++ myrivet.cc -o myrivet \
  `rivet-config --cppflags --ldflags --libs` \
  `agile-config --cppflags --ldflags --libs`
```

would be needed to compile the following AGILe+Rivet code:

```
#include "AGILe/Loader.hh"
#include "AGILe/Generator.hh"
#include "Rivet/AnalysisHandler.hh"
#include "HepMC/GenEvent.h"
#include "HepMC/IO_GenEvent.h"

using namespace std;

int main() {
  // Have a look what generators are available
  AGILe::Loader::initialize();
  const vector<string> gens = AGILe::Loader::getAvailableGens();
  foreach (const string& gen, gens) {
    cout << gen << endl;
  }

  // Load libraries for a specific generator and instantiate it
  AGILe::Loader::loadGenLibs("Pythia6:425");
  AGILe::Generator* generator = AGILe::Loader::createGen();
  cout << "Running " << generator->getName()
```

```
      << " version " << generator->getVersion() << endl;

  // Set generator initial state for LEP
  const int particle1 = AGILe::ELECTRON;
  const int particle2 = AGILe::POSITRON;
  const double sqrts = 91;
  generator->setInitialState(particle1, energy1, sqrts/2.0, sqrts/2.0);
  generator->setSeed(14283);

  // Set some parameters
  generator->setParam("MSTP(5)", "320"); //< PYTHIA tune
  // ...

  // Set up Rivet with a LEP analysis
  Rivet::AnalysisHandler rivet;
  rivet.addAnalysis("DELPHI_1996_S3430090");

  // Run events
  const int EVTMAX = 10000;
  HepMC::GenEvent evt;
  for (int i = 0; i < EVTMAX; ++i) {
    generator->makeEvent(evt);
    rivet.analyze(evt);
  }

  // Finalize Rivet and generator
  rivet.finalize();
  rivet.writeData("out.aida");
  generator->finalize();

  return 0;
}
```

10. Conclusions

We have presented a users' guide for the Rivet event-generator validation system. This manual is intended to be a guide to using Rivet, rather than a comprehensive reference to the application programming interface (API) of the

Rivet library. Rivet is a C++ class library, which provides the infrastructure and calculational tools for simulation-level analyses for high energy collider experiments, enabling physicists to validate event generator models and tunings with minimal effort and maximum portability. It is designed to scale effectively to large numbers of analyses for truly global validation, by transparent use of an automated result caching system.

In addition to an introduction to the philosophy behind the framework, we have given examples on how to implement the user's own analysis module. A selected list of available analyses has been given as an example of the flexibility of the full framework.

Part IV
Appendices

Appendix A. Typical `agile-runmc` commands

- **Simple run:** `agile-runmc Herwig:6510 -P lep1.params --beams=LEP:91.2 \ -n 1000` will use the Fortran Herwig 6.5.10 generator (the `-g` option switch) to generate 1000 events (the `-n` switch) in LEP1 mode, i.e. e^+e^- collisions at $\sqrt{s} = 91.2$ GeV.

- **Parameter changes:** `agile-runmc Pythia6:425 --beams=LEP:91.2 \ -n 1000 -P myrun.params -p "PARJ(82)=5.27"` will generate 1000 events using the Fortran Pythia 6.423 generator, again in LEP1 mode. The `-P` switch is actually the way of specifying a parameters file, with one parameter per line in the format "⟨*key*⟩ ⟨*value*⟩": in this case, the file `lep1.params` is loaded from the ⟨*installdir*⟩`/share/AGILe` directory, if it isn't first found in the current directory. The `-p` (lower-case) switch is used to change a named generator parameter, here Pythia's `PARJ(82)`, which sets the parton shower cutoff scale. Being able to change parameters on the command line is useful for scanning parameter ranges from a shell loop, or rapid testing of parameter values without needing to write a parameters file for use with `-P`.

- **Writing out HepMC events:** `agile-runmc Pythia6:425 --beams=LHC:14TeV -n 50 -o out.hepmc -R` will generate 50 LHC events with Pythia. The `-o` switch is being used here to tell `agile-runmc` to write the generated events to the `out.hepmc` file. This file will be a plain text dump of the HepMC event records in the standard HepMC format. Use of filename "-" will result in the event stream being written to standard output (i.e. dumping to the terminal.

Appendix B. Acknowledgements

Rivet development has been supported by a variety of sources:

- All authors acknowledge support from the EU MCnet research network. MCnet is a Marie Curie Training Network funded under Framework Programme 6 contract MRTN-CT-2006-035606 and Framework Programme 7 contract PITN-GA-2012-315877.

- Andy Buckley has been supported by grants from the UK Science and Technology Facilities Council (Special Project Grant), the Scottish Universities Physics Alliance (Advanced Research Fellowship), the Institute for Particle Physics Phenomenology (Associateship), and a CERN Scientific Associateship.

- Holger Schulz and Frank Siegert acknowledge the support of the German Research Foundation (DFG).

We also wish to thank the CERN MCplots (`http://mcplots.cern.ch`) team, and especially Anton Karneyeu, for doing the pre-release testing since the Rivet 1.5 series and pointing out all the bits that we got wrong: Rivet is a much better system as a result!

References

[1] M. Dobbs and J. B. Hansen, The HepMC C++ Monte Carlo event record for High Energy Physics, Comput. Phys. Commun. 134 (2001) 41–46. doi:10.1016/S0010-4655(00)00189-2.

[2] M. R. Whalley, D. Bourilkov, R. C. Group, The Les Houches Accord PDFs (LHAPDF) and LhagluearXiv:hep-ph/0508110.

[3] M. Cacciari, G. P. Salam, Dispelling the N^3 myth for the k_t jet-finder, Phys. Lett. B641 (2006) 57–61. arXiv:hep-ph/0512210, doi:10.1016/j.physletb.2006.08.037.

[4] M. Cacciari and G. Salam and G.Soyez, FastJet web site-Http://www.fastjet.fr.

[5] T. Sjostrand, S. Mrenna, P. Skands, PYTHIA 6.4 physics and manual, JHEP 05 (2006) 026. arXiv:hep-ph/0603175.

[6] T. Gleisberg, S. Hoeche, F. Krauss, M. Schonherr, S. Schumann, et al., Event generation with Sherpa 1.1, JHEP 0902 (2009) 007. arXiv:0811.4622, doi:10.1088/1126-6708/2009/02/007.

[7] T. Sjostrand, S. Mrenna, P. Skands, A Brief Introduction to PYTHIA 8.1, Comput. Phys. Commun. 178 (2008) 852–867. arXiv:0710.3820, doi:10.1016/j.cpc.2008.01.036.

[8] T. Sjostrand, Pythia 8 Status ReportarXiv:0809.0303.

[9] M. Bahr, et al., Herwig++ Physics and Manual, Eur. Phys. J. C58 (2008) 639–707. arXiv:0803.0883, doi:10.1140/epjc/s10052-008-0798-9.

[10] P. Abreu, et al., Tuning and test of fragmentation models based on identified particles and precision event shape data, Z. Phys. C73 (1996) 11–60. doi:10.1007/s002880050295.

[11] I. Antcheva, et al., ROOT: A C++ framework for petabyte data storage, statistical analysis and visualization, Comput. Phys. Commun. 180 (2009) 2499–2512. doi:10.1016/j.cpc.2009.08.005.

[12] D. Buskulic, et al., First measurement of the quark to photon fragmentation function, Z. Phys. C69 (1996) 365–378. doi:10.1007/s002880050037.

[13] K. Aamodt, et al., Production of pions, kaons and protons in pp collisions at $\sqrt{s} = 900$ GeV with ALICE at the LHC., Eur.Phys.J. C 71 (2011) 1655. arXiv:1101.4110, doi:10.1140/epjc/s10052-011-1655-9.

[14] H. Albrecht, et al., Inclusive production of charged pions, kaons and protons in upsilon (4S) decays, Z.Phys. C58 (1993) 191–198. doi:10.1007/BF01560337.

[15] G. Aad, et al., Measurement of $t\bar{t}$ production with a veto on additional central jet activity in pp collisions at $\sqrt{s} = 7$ TeV using the ATLAS detector, Eur.Phys.J. C72 (2012) 2043. arXiv:1203.5015, doi:10.1140/epjc/s10052-012-2043-9.

[16] B. Aubert, et al., Exclusive branching fraction measurements of semileptonic tau decays into three charged hadrons, $\tau^- \to \phi\pi^-\nu_\tau$ and $\tau^- \to \phi K^-\nu_\tau$, Phys. Rev. Lett. 100 (2008) 011801. arXiv:0707.2981, doi:10.1103/PhysRevLett.100.011801.

[17] R. Seuster, et al., Charm hadrons from fragmentation and B decays in e^+e^- annihilation at $\sqrt{s} = 10.6\,\text{GeV}$, Phys. Rev. D73 (2006) 032002. arXiv:hep-ex/0506068, doi:10.1103/PhysRevD.73.032002.

[18] T. Affolder, et al., Charged jet evolution and the underlying event in $p\bar{p}$ collisions at 1.8 TeV, Phys. Rev. D65 (2002) 092002. doi:10.1103/PhysRevD.65.092002.

[19] M. Artuso, et al., Charm meson spectra in $e + e-$ annihilation at 10.5 GeV, Phys. Rev. D70 (2004) 112001. arXiv:hep-ex/0402040, doi:10.1103/PhysRevD.70.112001.

[20] V. Khachatryan, et al., First Measurement of Hadronic Event Shapes in pp Collisions at $\sqrt{s} = 7$ TeV, Phys. Lett. B699 (2011) 48–67. arXiv:1102.0068, doi:10.1016/j.physletb.2011.03.060.

[21] V. M. Abazov, et al., Measurement of the differential cross-section for the production of an isolated photon with associated jet in $p\bar{p}$ collisions

at $\sqrt{s} = 1.96$ TeV, Phys. Lett. B666 (2008) 435–445. arXiv:0804.1107, doi:10.1016/j.physletb.2008.06.076.

[22] C. Adloff, et al., Measurements of transverse energy flow in deep inelastic-scattering at HERA, Eur. Phys. J. C12 (2000) 595–607. arXiv:hep-ex/9907027, doi:10.1007/s100520000287.

[23] P. A. Movilla Fernandez, O. Biebel, S. Bethke, S. Kluth, P. Pfeifenschneider, A study of event shapes and determinations of α_s using data of e^+e^- annihilations at $\sqrt{s} = 22$ GeV to 44 GeV, Eur. Phys. J. C1 (1998) 461–478. arXiv:hep-ex/9708034, doi:10.1007/s100520050096.

[24] R. Aaij, et al., Measurement of the inclusive ϕ cross-section in pp collisions at $\sqrt{s} = 7$ TeV, Phys.Lett. B703 (2011) 267–273. arXiv:1107.3935, doi:10.1016/j.physletb.2011.08.017.

[25] O. Adriani, et al., Measurement of forward neutral pion transverse momentum spectra for $\sqrt{s} = 7$ TeV proton-proton collisions at LHC, Phys.Rev. D86 (2012) 092001. arXiv:1205.4578, doi:10.1103/PhysRevD.86.092001.

[26] G. Abbiendi, et al., Measurement of event shape distributions and moments in $e^+e^- \to$ hadrons at 91 GeV - 209 GeV and a determination of α_s, Eur. Phys. J. C40 (2005) 287–316. arXiv:hep-ex/0503051, doi:10.1140/epjc/s2005-02120-6.

[27] C. Amsler, et al., Review of particle physics, Phys. Lett. B667 (2008) 1. doi:10.1016/j.physletb.2008.07.018.

[28] K. Abe, et al., Production of π^+, π^-, K^+, K^-, p and $\bar{p}$ in light (uds), c and b jets from Z^0 decays, Phys. Rev. D69 (2004) 072003. arXiv:hep-ex/0310017, doi:10.1103/PhysRevD.69.072003.

[29] J. Adams, et al., Identified hadron spectra at large transverse momentum in $p\,p$ and $d\,Au$ collisions at $\sqrt{s} = 200$ GeV, Phys. Lett. B637 (2006) 161–169. arXiv:nucl-ex/0601033, doi:10.1016/j.physletb.2006.04.032.

[30] W. Braunschweig, et al., Global jet properties at 14 GeV to 44 GeV center-of-mass energy in e^+e^- annihilation, Z. Phys. C47 (1990) 187–198. doi:10.1007/BF01552339.

[31] P. Aspell, Measurement of the forward charged particle pseudorapidity density in pp collisions at $\sqrt{s} = 7$ TeV with the TOTEM experiment, Europhys.Lett. 98 (2012) 31002. arXiv:1205.4105, doi:10.1209/0295-5075/98/31002.

[32] C. Albajar, et al., A Study of the General Characteristics of $p\bar{p}$ Collisions at $\sqrt{s} = 0.2$ TeV to 0.9 TeV, Nucl. Phys. B335 (1990) 261. doi:10.1016/0550-3213(90)90493-W.

[33] K. Alpgard, et al., Comparison of $p\bar{p}$ and pp interactions at $\sqrt{s} = 53$ GeV, Phys. Lett. B112 (1982) 183. doi:10.1016/0370-2693(82)90325-2.

[34] J. Bromley, et al., HZTool: A package for Monte Carlo-data comparison at HERA (version 1.0)ZEUS and H1 Collaborations.